# PINN Code Documentation

Ameir Shaa

November 10, 2023

## Contents

## 1 Introduction

This documentation provides a comprehensive overview of the utility functions and classes present in the code, offering insights into their purpose and utility in the broader context of deep learning model development and evaluation.

## 2 Configuration Script Documentation

**Overview:**

- This script contains configurations for various components of the application.
- These configurations are used to define and modify behavior for optimizers, training setups, distributed training, machine-specific paths, plotting options, data processing, and loss components.
- The script also contains a main function which sets up the directory structure based on the chosen machine and invokes the main function from the `main` module.

**Configuration Details:**

```python
from main import *
import os
import numpy as np

# Configuration dictionary containing all the setup details.
config = {
    # Configurations for LBFGS optimizer.
    "lbfgs_optimizer": {...},

    # Configurations for Adam optimizer.
    "adam_optimizer": {...},

    # Configurations when using both LBFGS and Adam optimizers.
    "both_optimizers": {...},

    # General training configurations.
    "training": {...},

    # Configurations for distributed training setup.
    "distributed_training": {...},

    # Paths for different machines.
    "machine": {...},

    # Plotting configurations.
    "plotting": {...},

    # Data related configurations.
    "data": {...},

    # Configurations related to different loss components.
    "loss_components": {...},

    # Train-test configurations.
    "train_test": {...},

    # Machine and optimizer choice.
    "chosen_machine": "mac",
    "chosen_optimizer": "adam_optimizer",

    # Base folder name.
    "base_folder_name": '26102023_adam_datalossonly_infinite'
}

# Main block that sets up directories and invokes the main function.
if __name__ == "__main__":
    ...
```

## 3 Main Script Documentation

**Overview:**

- This script provides the main execution workflow for the application.
- It integrates various functionalities such as data loading, training, testing, evaluation, and plotting.
- The configurations provided from the `config.py` are utilized to drive the behavior and settings of the execution.

**Imports:**

- `definitions`: Contains various constant definitions and utility functions.
- `training`: Contains functions and logic related to training the model.
- `PINN`: Defines the Physics Informed Neural Network (PINN) architecture and related operations.
- `plotting`: Provides functionality to plot and visualize data and model predictions.

**Function** `main(base_directory, config, output_zip_file=None)`**:**

- **Parameters:**
  - `base_directory`: The root directory where output and logs will be saved.
  - `config`: Configuration dictionary from `config.py` which dictates the behavior of the execution.
  - `output_zip_file (Optional)`: If provided, the output will be zipped and saved to this file.

- **Workflow:**
  1. Setup directories for logs and outputs.
  2. Load data and preprocess it based on configurations.
  3. (Optional) Plot pure data without model predictions.
  4. Initialize the PINN model.
  5. Train the model based on configurations.
  6. Test and evaluate the model on test data.
  7. (Optional) Evaluate the model on special test data (e.g., skipped angles).
  8. (Optional) Zip and save all outputs to a provided zip file.

```python
from definitions import *
from training import *
from PINN import *
from plotting import *

def main(base_directory, config, output_zip_file=None):
    ...
```

## 4 PINN Initialization Documentation

**Overview:**

- The `PINN` class defines the architecture and functionalities of the Physics-Informed Neural Network.
- This class inherits from the `nn.Module` class provided by PyTorch.
- The network consists of several fully connected layers and provides methods for forward propagation with different activation functions.
- It also includes methods to compute different losses, such as data loss, boundary loss, and physics-informed losses.

**Network Architecture:**

- Input Layer: 5 neurons.
- Hidden Layers: Four layers each with 128 neurons.
- Output Layer: 5 neurons.

**Methods:**

- `forward_relu(self, x)`: Propagates the input through the network using the ReLU activation function.

- `forward_tanh(self, x)`: Propagates the input through the network using the Tanh activation function.
- `forward(self, x, activation)`: General forward method that decides which activation function to use based on the provided argument.
- `compute_data_loss(self, X, y, activation)`: Computes the mean squared error between the model's predictions and the true values.
- `compute_boundary_loss(self, X_boundary, y_boundary_known, activation)`: Computes the boundary loss based on the velocity components.
- `extract_parameters(self, X)`: Extracts specific parameters from the input tensor.
- `RANS(self, wind_angle, x, y, z, rho, nu, activation)`: Defines the RANS equations and computes various gradients and terms based on the model's predictions.
- `compute_physics_cont_loss(self, X)`: Computes the physics-informed continuity loss.

```python
class PINN(nn.Module):
    def __init__(self):
        ...
    def forward_relu(self, x):
        ...
    def forward_tanh(self, x):
        ...
    def forward(self, x, activation):
        ...
    def compute_data_loss(self, X, y, activation):
        ...
    def compute_boundary_loss(self, X_boundary, y_boundary_known,
   activation):
        ...
    def extract_parameters(self, X):
        ...
    def RANS(self, wind_angle, x, y, z, rho, nu, activation):
        ...
    def compute_physics_cont_loss(self, X):
        ...
```

## 4.1 RANS

### 4.1.1 Introduction to RANS Equations

Turbulent flows are characterized by irregular fluctuations and a high degree of mixing. Due to the complex nature of turbulence, solving the Navier-Stokes equations directly for each point in space and time (a method known as Direct Numerical Simulation or DNS) requires an enormous computational effort, which is impractical for most engineering applications. As a compromise, engineers and scientists use time-averaged equations for turbulent flows, which give rise to the Reynolds-Averaged Navier-Stokes (RANS) equations.

The RANS approach simplifies the problem by decomposing the instantaneous flow variables into mean and fluctuating components. The mean components represent the average flow field, while the fluctuating components account for the turbulent fluctuations around the mean. By averaging the Navier-Stokes equations, the RANS equations incorporate the effects of turbulence into additional stress terms, known as Reynolds stresses, which must be modeled to close the system of equations.

### 4.1.2 Reynolds Decomposition

Reynolds decomposition separates the instantaneous velocity and pressure fields into their mean and fluctuating components. If $U_i$ represents the instantaneous velocity field and $\overline{U}_i$ its time-averaged counterpart, the fluctuating component $U_i'$ is defined as:

$$U_i = \overline{U}_i + U_i' \tag{1}$$

Similarly, for the pressure field $p$, we have:

$$p = \overline{p} + p' \tag{2}$$

The overbar denotes time-averaging, and the prime denotes the fluctuating component. This decomposition allows for the separation of the effects of the turbulent fluctuations from the mean flow in the Navier-Stokes equations.

### 4.1.3 Derivation of RANS Equations

Starting from the Navier-Stokes equations, the RANS equations are derived by taking the time-average of the equations, leading to additional terms involving the products of fluctuating components. The resulting equations contain terms called the Reynolds stresses, which represent the effect of the turbulence on the mean flow. The RANS equations take the following general form for the continuity equation for incompressible flow and conservation of momentum respectively:

The continuity equation as well as the python excerpt for incompressible flow is given by:

$$\frac{\partial \overline{U}_i}{\partial x_i} = 0 \tag{3}$$

```
    cont = u_x + v_y + w_z
```

The RANS momentum equations are expressed as:

$$\rho \left( \frac{\partial \overline{U}_i}{\partial t} + \overline{U}_j \frac{\partial \overline{U}_i}{\partial x_j} \right) = -\frac{\partial \overline{p}}{\partial x_i} + \nu \frac{\partial^2 \overline{U}_i}{\partial x_j^2} - \frac{\partial \overline{u_i' u_j'}}{\partial x_j} \tag{4}$$

where $\rho$ is the fluid density, $\nu$ is the dynamic viscosity, $\overline{U}_i$ are the mean velocity components, $\overline{p}$ is the mean pressure, and $\overline{u_i'u_j'}$ are the Reynolds stresses. The challenge in solving the RANS equations lies in modeling these Reynolds stresses.

The RANS momentum equations in component form as well as the python excerpt for incompressible flow are given by

$$\rho\left(\frac{\partial\overline{u}}{\partial t}+\overline{u}\frac{\partial\overline{u}}{\partial x}+\overline{v}\frac{\partial\overline{u}}{\partial y}+\overline{w}\frac{\partial\overline{u}}{\partial z}\right)=-\frac{\partial\overline{p}}{\partial x}+\nu\left(\frac{\partial^2\overline{u}}{\partial x^2}+\frac{\partial^2\overline{u}}{\partial y^2}+\frac{\partial^2\overline{u}}{\partial z^2}\right)-\rho\left(\frac{\partial\overline{u'v'}}{\partial y}+\frac{\partial\overline{u'w'}}{\partial z}\right) \tag{5}$$

```
    f = rho * (u * u_x + v * u_y + w * u_z) + p_x - nu * (u_xx + u_yy +
u_zz) - stress_tensor_x[0] - stress_tensor_y[0] - stress_tensor_z[0]
```

$$\rho\left(\frac{\partial\overline{v}}{\partial t}+\overline{u}\frac{\partial\overline{v}}{\partial x}+\overline{v}\frac{\partial\overline{v}}{\partial y}+\overline{w}\frac{\partial\overline{v}}{\partial z}\right)=-\frac{\partial\overline{p}}{\partial y}+\nu\left(\frac{\partial^2\overline{v}}{\partial x^2}+\frac{\partial^2\overline{v}}{\partial y^2}+\frac{\partial^2\overline{v}}{\partial z^2}\right)-\rho\left(\frac{\partial\overline{u'v'}}{\partial x}+\frac{\partial\overline{v'w'}}{\partial z}\right) \tag{6}$$

```
    g = rho * (u * v_x + v * v_y + w * v_z) + p_y - nu * (v_xx + v_yy +
v_zz) - stress_tensor_x[1] - stress_tensor_y[1] - stress_tensor_z[1]
```

$$\rho\left(\frac{\partial\overline{w}}{\partial t}+\overline{u}\frac{\partial\overline{w}}{\partial x}+\overline{v}\frac{\partial\overline{w}}{\partial y}+\overline{w}\frac{\partial\overline{w}}{\partial z}\right)=-\frac{\partial\overline{p}}{\partial z}+\nu\left(\frac{\partial^2\overline{w}}{\partial x^2}+\frac{\partial^2\overline{w}}{\partial y^2}+\frac{\partial^2\overline{w}}{\partial z^2}\right)-\rho\left(\frac{\partial\overline{u'w'}}{\partial x}+\frac{\partial\overline{v'w'}}{\partial y}\right) \tag{7}$$

```
    h = rho * (u * w_x + v * w_y + w * w_z) + p_z - nu * (w_xx + w_yy +
w_zz) - stress_tensor_x[2] - stress_tensor_y[2] - stress_tensor_z[2]
```

### 4.1.4 Reynolds Stress Tensor in the $k-\epsilon$ Model

Using the computed turbulent viscosity $\nu_t$, the Reynolds stress tensor can be formulated. The deviatoric Reynolds stress tensor $\tau_{ij}^{'}$ is modeled as:

$$\tau_{ij}^{'}=-2\nu_t S_{ij}+\frac{2}{3}k\delta_{ij} \tag{8}$$

where $S_{ij}$ is the mean rate of strain tensor defined by:

$$S_{ij}=\frac{1}{2}\left(\frac{\partial U_i}{\partial x_j}+\frac{\partial U_j}{\partial x_i}\right) \tag{9}$$

This tensor represents the effect of turbulence on the mean flow and is used in the RANS equations to account for the turbulent stresses. Below is the equivalent Python excerpt from the code:

```python
import torch

# Constants for the $k-\epsilon$ model
C_mu = 0.09

def compute_reynolds_stress(nu_t, u, v, w, k, epsilon, u_grad, v_grad,
    w_grad, rho):
    % # Calculate turbulent viscosity nu_t
    % nu_t = rho * C_mu * (k**2 / epsilon)

    # Calculate the mean rate of strain tensor components
    S_xx = 0.5 * (u_grad[:, 0] + u_grad[:, 0])
    S_yy = 0.5 * (v_grad[:, 1] + v_grad[:, 1])
    S_zz = 0.5 * (w_grad[:, 2] + w_grad[:, 2])
    S_xy = 0.5 * (u_grad[:, 1] + v_grad[:, 0])
    S_xz = 0.5 * (u_grad[:, 2] + w_grad[:, 0])
    S_yz = 0.5 * (v_grad[:, 2] + w_grad[:, 1])

    # Initialize the Reynolds stress tensor
    stress_tensor = torch.zeros(u_grad.shape[0], 3, 3, dtype=u.dtype,
    device=u.device)

    # Compute the Reynolds stress tensor using the Boussinesq hypothesis
    stress_tensor[:, 0, 0] = -2 * nu_t * S_xx + (2/3) * k
    stress_tensor[:, 1, 1] = -2 * nu_t * S_yy + (2/3) * k
    stress_tensor[:, 2, 2] = -2 * nu_t * S_zz + (2/3) * k
    stress_tensor[:, 0, 1] = stress_tensor[:, 1, 0] = -nu_t * S_xy
    stress_tensor[:, 0, 2] = stress_tensor[:, 2, 0] = -nu_t * S_xz
    stress_tensor[:, 1, 2] = stress_tensor[:, 2, 1] = -nu_t * S_yz

    return stress_tensor

# Example usage:
# Assume u, v, w, k, epsilon, u_grad, v_grad, w_grad, rho are defined
    tensors. For example, u_grad is a tensor with shape (N, 3) where N is
    the number of points and contains the gradients [du/dx, du/dy, du/dz]
    for each point. Then,
stress_tensor = compute_reynolds_stress(nu_t, u, v, w, k, epsilon, u_grad,
     v_grad, w_grad, rho)
```

### 4.1.5 CFD Implementation

Implementing the $k - \epsilon$ model within a computational fluid dynamics (CFD) code involves several steps. First, the governing equations (RANS and $k-\epsilon$ equations) are discretized using appropriate numerical methods. Then, boundary conditions are applied, and the equations are solved iteratively to obtain a converged solution. The $k - \epsilon$ model is integrated into the solver, and $\nu_t$ is calculated at each iteration. This turbulent viscosity is then used to compute the Reynolds stresses, which are included in the momentum equations.

#### 4.1.6 Conclusion

The RANS equations with the $k - \epsilon$ turbulence model provide a practical approach to simulating turbulent flows in engineering and scientific applications. Although the model has limitations and may not predict all features of turbulence accurately, it is a well-established and widely used method in the field of CFD. Further research and development continue to improve turbulence models and enhance their accuracy and applicability.

# 5 PINN Training Explanation

The function `train_model` provides the training mechanism for a given deep learning model. It has been designed to handle both distributed and non-distributed training sessions, based on the provided configuration. Specifically, the function is structured to:

1. Set up the distributed training environment, if specified.
2. Load the model onto the appropriate device.
3. Define the optimization strategy.
4. Define functions for computing losses.
5. Calculate optimal batch sizes, if batch training is desired.
6. Iterate through epochs and batches, updating the model's parameters using the defined optimization strategy.
7. Periodically save the model's state to a file.
8. Return the trained model.

## 5.1 Detailed Explanation

### 5.1.1 Distributed Training Initialization

If distributed training is specified in the `config`, the function initializes the distributed training environment using the `init_process_group` function from PyTorch's distributed package. The IP address, port number, and backend are all fetched from the configuration.

### 5.1.2 Model Device Allocation

The model is then loaded onto the specified device. If GPU is available and distributed training is enabled, it wraps the model in `DistributedDataParallel` to ensure that the model can be trained on multiple GPUs.

#### 5.1.3 Loss Function Definitions

Two functions are defined within `train_model` to compute different losses:

- `total_boundary_loss`: Computes the boundary loss for the model. It sums the losses from the no-slip condition and the inlet velocity condition for different wind angles.
- `calculate_total_loss`: Calculates the total loss by combining various losses such as data loss, momentum loss, continuity loss, and the above-defined boundary loss.

#### 5.1.4 Batch Size Calculation

If batch training is specified, the function calculates the optimal batch size for the given model and data. This is done iteratively, starting from a specified batch size and increasing it until the GPU memory limits are reached.

#### 5.1.5 Training Loop

The main training loop iterates through the epochs and, if batches are used, through the batches within each epoch. Within the loop, the optimizer updates the model's parameters based on the computed gradients from the total loss.

#### 5.1.6 Model Checkpointing

Every few epochs (as specified by the modulo operation `epoch % 5 == 0`), the model's state (including its parameters and the optimizer's state) is saved to a file. This allows for recovery in case training is interrupted. If the model file already exists, the function can also resume training from the last saved checkpoint.

#### 5.1.7 Stopping Criteria

The function uses both a defined number of epochs and an early stopping mechanism based on the difference in loss between consecutive epochs. If the loss doesn't change significantly (below a defined threshold) for a specified number of consecutive epochs, training is stopped.

#### 5.1.8 Model Return

Finally, after the training loop is completed, the function saves the final state of the model and returns the trained model.

## 5.2 Conclusion

This function appears to be a comprehensive training routine, integrating best practices such as early stopping, model checkpointing, and distributed training. It's designed to be versatile and to handle different training scenarios and configurations.

# 6 Weighting Documentation

The `weighting.py` script contains a single function, `weighting`, that computes the weighted total loss for the model being trained.

## 6.1 Breakdown

1. **Imports**:
    - It imports some necessary components from the `definitions.py` module.
2. **Function**:
    - **Name**: `weighting`
    - **Inputs**: The function takes four arguments:
        - `data_loss`: Loss due to the discrepancy between predicted and actual data.
        - `total_averaged_boundary_loss`: The average loss calculated from the boundary conditions.
        - `cont_loss`: Loss from the continuity equation (physics-based).
        - `momentum_loss`: Loss from the momentum equation (physics-based).
    - **Description**:
        - The function first calculates two cumulative losses:
            - `data_boundary_loss`: This is the sum of the data loss and the boundary loss.
            - `all_physics_loss`: This is the sum of the continuity and momentum losses.
        - Next, it calculates the weights for the data-boundary and physics-based losses:
            - `weight_data_boundary`: This is the fraction of the `data_boundary_loss` in the sum of `data_boundary_loss` and `all_physics_loss`.
            - `weight_physics`: This is the fraction of the `all_physics_loss` in the sum of `data_boundary_loss` and `all_physics_loss`.
        - It then calculates the `total_loss` as the weighted sum of the data-boundary and physics-based losses.
    - **Output**: The function returns the `total_loss`.

The purpose of this function is to balance the contributions of data-based loss (arising from training data) and physics-based losses (arising from physical equations) so that neither dominates the training process. The model is trained to fit both the provided data and the known physical laws.

# 7 Boundary Conditions Documentation

The `boundary.py` module is dedicated to defining and handling boundary conditions, specifically for a given domain that includes a sphere and a cylinder. The module defines various types of boundary conditions such as Dirichlet, Neumann, slip/no-slip conditions, and inflow/outflow conditions. Additionally, the file contains functions to sample points from the sphere, cylinder, and domain boundaries and to enforce the no-slip boundary conditions on the geometries.

## 7.1 Boundary Condition Types

### 7.1.1 Dirichlet Boundary Conditions

These specify the exact value of the solution at the boundary. An example use-case could be when one knows the exact wind velocity at the domain boundaries.

### 7.1.2 Neumann Boundary Conditions

These conditions specify the derivative of the solution at the boundary. In other words, this condition would be used if there's information about how a quantity, like wind velocity, changes as one approaches the domain boundaries.

### 7.1.3 Slip/No-Slip Conditions

For surfaces like those of the sphere and cylinder, the module considers two possibilities:

- **No-Slip Conditions**: Here, the velocity is zero, indicating that the fluid does not move along the surface.
- **Slip Conditions**: The velocity is not necessarily zero, but its normal component (perpendicular to the surface) is zero.

### 7.1.4 Inflow/Outflow Conditions

This pertains to specifying which boundaries of the domain have fluid (or wind) entering (inflow) or exiting (outflow).

### 7.1.5 Initial Conditions

If dealing with a time-dependent problem, initial conditions specify the state of the system at the starting time.

## 7.2 Functions

### 7.2.1 parameters

Returns a collection of parameters related to the sphere, cylinder, domain boundaries, and wind angles. Parameters include dimensions, number of sample points, and velocity values.

#### 7.2.2 sample_sphere_surface

Given the center, radius, number of points, and wind direction, this function samples points from the surface of a sphere. The sampling is based on spherical coordinates.

#### 7.2.3 sample_cylinder_surface

This function works similarly to the sphere sampling function, but samples points from the surface of a cylinder. It separately samples points from the lateral surface and the rounded cap of the cylinder.

#### 7.2.4 sample_domain_boundary

Given the domain boundaries and the wind direction, this function samples points from the boundary and computes the x, y, and z components of the inlet velocity based on the given wind direction.

#### 7.2.5 no_slip_boundary_conditions

This function enforces the no-slip boundary condition on the sphere and cylinder surfaces. It samples points from both surfaces and assigns zero velocities to them, representing the no-slip condition.

### 7.3 Conclusion

The `boundary.py` module provides a comprehensive set of tools and definitions to handle boundary conditions for a given problem domain that includes a sphere and a cylinder. It ensures that the boundaries are treated correctly based on the specified conditions, allowing for accurate simulations and analyses.

# 8 Plotting Script Documentation

The `plotting.py` script is designed to provide functions for plotting and visualizing data in various formats. This documentation covers the functionality of the script, detailing the purpose and usage of each function. The second part of the script provides functions for plotting and visualizing the data. These functions enable users to obtain insights from their data through visual representations.

## 8.1 Part 1: Data Loading and Preprocessing

In the first part of the script, there are functions dedicated to loading and preprocessing data. These functions ensure that data is loaded correctly and prepared for visualization.

### 8.1.1 load_data(filename)

```python
def load_data(filename):
    # Load data from the given filename
    pass
```

This function is designed to load data from a specified file. Depending on the implementation, various file formats such as CSV, Excel, or databases can be supported.

### 8.1.2 preprocess_data(data)

```python
def preprocess_data(data):
    # Preprocess the given data
    pass
```

After data is loaded, it might require some preprocessing before it can be visualized. This function provides a placeholder for such preprocessing tasks. It could include tasks such as cleaning the data, handling missing values, or transforming the data into a desirable format.

## 8.2 Plotting and Visualization

This document provides a detailed breakdown of the `plot_data_predictions` function used to compare the actual and predicted values of velocity and pressure for different planes in a dataset.

## 8.3 Function Overview

The function `plot_data_predictions` is designed to facilitate the visual comparison between actual and predicted values of velocity and pressure across different planes. The function operates in several steps as highlighted below:

1. Determine the plane of interest based on user input.
2. Filter data based on the given wind angle and a specific tolerance.
3. Creation of a regular grid over the selected plane.
4. Interpolation of actual and predicted values onto the grid.
5. Visualization of a geometric structure, perhaps representing an experimental setup.
6. Generation of contour plots for both actual and predicted values.
7. Saving the generated plots as image files.

## 8.4 Detailed Explanation

### 8.4.1 Determine Plane of Interest

The function starts by determining the plane of interest based on user input. Depending on the plane (X-Z, Y-Z, or X-Y), the function sets the appropriate axes for plotting and the axis for filtering the data.

### 8.4.2 Filter Data

Data is filtered based on the given wind angle. An additional filter is applied to the data based on a specified tolerance around the cut value for the third axis.

### 8.4.3 Grid Creation and Interpolation

A regular grid is created over the range of the two axes of interest. The actual and predicted values of velocities and pressure are then interpolated onto this grid.

### 8.4.4 Geometry Fill

The function visualizes a geometric structure, which seems to be a combination of a sphere and a cylinder. This could be indicative of the physical setup or experimental design.

### 8.4.5 Plotting

The function generates several subplots, each comparing the actual and predicted values for:

- Total velocity magnitude
- X-component of velocity
- Y-component of velocity
- Z-component of velocity
- Pressure

Each subplot consists of two contour plots placed side-by-side; one showcases the actual values while the other depicts the predicted values. Color bars are also added to provide a scale for the contour plots.

### 8.4.6 Saving Plots

The generated subplots are saved as image files with names provided as arguments to the function.

## 8.5 Suggestions

A significant portion of the plotting subsection appears repetitive. It's advisable to further modularize the code by creating helper functions for repetitive tasks. This would not only make the code more concise but also enhance its maintainability.

## 8.6 Conclusion

The `plotting.py` script offers a basic structure for loading, preprocessing, and visualizing data. While the provided functions are placeholders, they can be expanded upon to support various data sources, preprocessing tasks, and visualization techniques.

# 9 Utility Functions and Classes

This document provides an in-depth explanation of the utility functions and classes present in the provided code. These utilities address several aspects of deep learning model development and evaluation, including memory management, data preprocessing, optimization, and model evaluation.

## 9.1 Logger Class

The **Logger** class is a utility that aids in monitoring the runtime behavior of the program. It is designed to capture and record both standard output messages and save them in a log file.

```
class Logger(object):
    ...
```

By redirecting the standard output, the Logger class ensures that important runtime information, such as errors, warnings, or other print statements, are not only displayed on the screen but also stored persistently. This can be particularly useful for long-running processes where manual monitoring might not be feasible.

## 9.2 Device Management and Memory Utilization

Deep learning, particularly with neural networks, often requires significant computational resources. The functions in this subsection are designed to manage and efficiently utilize the available computational resources.

### 9.2.1 Device Memory

**get_available_device_memory(device)** returns the available memory on a specified device, which can be either a GPU or CPU. This is especially useful when managing memory usage on GPUs, ensuring that data or models do not exceed the GPU's capacity.

**print_and_set_available_gpus()** detects and lists the available GPUs. It then selects the GPU with the most free memory for computational tasks. This function is a cornerstone for systems with multiple GPUs, ensuring optimal utilization.

### 9.2.2 Memory Estimation

**estimate_memory(model, input_tensor, batch_size)** provides a rough estimate of the memory requirement for a model's forward and backward pass. This is useful for dynamically setting batch sizes based on available memory, ensuring efficient training without memory overflows.

**select_device_and_batch_size(model, input_tensor, device)** uses the aforementioned memory estimation function to determine the most suitable batch size for training on the selected device.

## 9.3 Data Management

The code provides several utilities for managing and preprocessing data, which are crucial steps in any machine learning pipeline.

### 9.3.1 Data Loading and Preprocessing

**load_data(filenames, base_directory, datafolder_path, device, config)** handles data loading from given filenames. It also preprocesses the data by adding features such as wind angles, splits the data into training and test sets, and scales the features and targets.

### 9.3.2 Scaling

Data scaling is an essential preprocessing step in neural network training. The functions **initialize_and_fit_scalers()** and **transform_data_with_scalers()** handle the initialization and application of data scalers, ensuring that the data is appropriately normalized for training.

## 9.4 Model Evaluation

Once a model is trained, evaluating its performance is crucial. The functions in this subsection facilitate this evaluation.

**evaluate_model()** and **evaluate_model_skipped()** assess a model's performance on test data. These functions compute various metrics such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and R-squared. Results, along with predictions, are saved for further analysis.

# 10 Conclusion

This documentation serves as a guide to the code's functionality and usage. It aims to provide clarity and understanding to developers and users.